# 性能 Performance

## 简介

GPU性能提升为着色器程序的编写带来了更大的灵活性，更容易写出冗长的着色器代码。然而程序变长也意味着执行得更慢。在大多数情况下，编译器会对着色器代码进行优化，但高层次的算法优化仍需要程序员完成。接下来会介绍一些着色器程序的优化方法。

## 使用标准库函数 Use the Standard Library

标准库函数在编译时会被转换成高度优化的汇编代码，其中许多函数被编译为可以在一个GPU时钟循环运行的单汇编指令，比如：sin函数、cos函数、lit函数、dot函数和exp函数。

尽管如此，有时你也可以自己编写更高效的函数版本来适应特殊情况，比如使用标准化立方体贴图来获取标准化向量。

## 充分利用统一参数 Take Advantage of Uniform Parameters

编写着色器时，应尽量避免对统一参数的单独计算。例如：当你将一个名为time的变量传递给像素着色器时，你最好不要在程序中计算sin(time)的值，而是将sin(time)的值作为统一参数传递给着色器。即使像素着色器还需要time做其他计算，传递两个统一参数的效率也远远高于在着色器程序中计算sin(time)。

## 使用顶点着色器与使用像素着色器的对比 Using Vertex Programs vs. Pixel Programs

### 顶点着色器与像素着色器的区别

像素着色器能够产生比顶点着色器更精确的图像。如果场景的镶嵌细节程度很低，顶点着色器通常会制造出面片化的粗糙图像，但像素着色并不是没有代价的：

* 像素着色器程序会为每个生成的像素执行一次，所以通常每绘制一帧，像素着色器程序就要执行上百万次；而顶点着色器程序每帧只需要执行上万次（在模型细节程度不高的情况下）。
* 此外，像素着色器程序对循环与分支语句的限制也高于顶点着色器，这种限制在早期的显卡中尤为明显。

### 顶点着色器与像素着色器的工作分配法则

因此，如何分配顶点与像素着色器程序的工作负载变得十分重要。一个简单的经验法则是：“使用顶点着色器执行基本变换与纹理坐标操作，使用像素着色器执行剩余的计算”。通常情况下这是一个良好的方法，因为它将更昂贵的数学操作放在了顶点着色器中。

有时，也许你想通过降低图像质量的方式来换取性能提升。在这种情况下，可以将更多的计算移至顶点着色器中，起到减少像素着色器程序长度的目的。这种方式带来的图像质量损失也许会远低于你预期，因为插值计算通常能够起到很好的效果，尤其是当顶点属性线性变化时，在顶点着色器中计算将不会对图像质量造成任何损失。

综上所述，决定何时使用顶点着色器代替像素着色器时，需要考虑的因素是像素着色器程序的参数是如何变化的：

* 如果参数变化缓慢（比如漫反射照明），使用顶点着色器计算是可接受的
* 如果参数变化快速（比如镜面照明），则需要使用像素着色器计算。

## 数据类型以及它们对性能的影响 Data Types and Their Impact on Performance

为了达到最优性能，我们需要在适用任意特定计算的情况下，尽量使用最小的数据类型。

### 定点类型 Fixed-point Type

适用于低精度计算，比如色彩计算。它们往往具有比浮点数类型快几倍的计算速度。

### 半长度类型 Half Type

适用于表示更高精度和更大范围的数据，但不要求尽可能的精确。半长度类型非常适合用来表示与图形相关的属性，比如色彩与法线。当需要对色彩进行高精度计算时，半长度类型通常是足够的。除了节省存储空间外，对半长度类型执行计算的速度也快于完整类型。

### 浮点类型 Floating-point Type

适用于需要最高精度与最大取值范围的场合。比如表示顶点坐标。

## 利用矢量化的优势 Take Advantage of Vectorization

与独立的标量相比，对矢量类型Vector Type执行计算的效率要高得多。对大多数操作来说，对一个矢量执行所耗费的时间与对一个标量执行是一样的。这表示你可以通过谨慎地使用矢量，将某些指令序列的执行速度提高四倍。HLSL语言的调和与隐式书写特征方便了标量到矢量的插入与抽取。

在这里举一个例子：在顶点着色器中，我们可以通过下面的代码计算例子的位置：

float4 pFinal = pInitial + vInitial \* t + 0.5 \* acceleration \* t \* t

上述的写法比下面独立计算每个成分的代码快了四倍：

float4 pFinal;

pFinal.x = pInitial.x + vInitial.x \* t + 0.5 \* acceleration.x \* t \* t

pFinal.y = pInitial.y + vInitial.y \* t + 0.5 \* acceleration.y \* t \* t

pFinal.z = pInitial.z + vInitial.z \* t + 0.5 \* acceleration.z \* t \* t

pFinal.w = pInitial.w + vInitial.w \* t + 0.5 \* acceleration.w \* t \* t

除了更高效以外，矢量化代码往往比非矢量化代码更加简洁与清晰。

## 使用纹理编码函数 Use Textures to Encode Functions

加快像素着色器运算速度的一种方法是通过纹理查找而不是执行算术运算来实现函数。

考虑一个函数，它以一个浮点数作为输入参数，并返回一个浮点数作为结果。如果输入范围可以控制在[0.0f, 1.0f]之间，就可以将输入值当作纹理坐标，将函数关系转化为一个1D纹理，查找的结果是一个像素颜色，即函数的结果。而这种方法也可以推广到二维、三维或者四维。

使用纹理编码函数有很多优点：

* 运行过程中不需要执行算术运算，为GPU节省了时间
* 通过GPU特有的纹理过滤硬件，可以自动平滑你的函数（在你期望的情况下），即通过使用很小的纹理，让硬件在纹理像素之间进行插值
* 纹理贴图允许使用更灵活的方式编码函数，而不必使用特定的公式或模式建模

## 自由使用混合与取反 Use Swizzling and Negation Freely

顶点着色器提供了没有任何性能损失的变量调和（将变量取值范围限制在[0, 1]之间）与取反操作。

在支持CineFX的GPU中，像素着色器也支持无消耗的调和与取反操作（即saturate函数与abs函数无消耗）。

## 只对必要的像素点着色 Shade Only the Pixels That You Must

对于复杂程序来说，像素着色器程序通常会成为应用程序的瓶颈。在这种情况下，不浪费GPU周期用于处理不可见的区域是明智的选择。一个很好的方法是在不着色的同时，启用深度测试绘制一次场景。我们称这种方法为“首先确定z”。这个过程结束后，帧缓存中的每个像素都是可视的最近表面上的像素。在接下来的绘制过程中，通过配置深度测试，只绘制满足帧缓存中深度值的像素。通过这种方法，像素着色器程序只为可见像素执行，而不是所有可能被看见的像素。

在D3D中，深度测试发生在图形流水线末端，此时像素着色器已经执行完毕。然而幸运的是，大多数现代图形处理器都提供了专门的硬件用于在像素着色器程序运行之前执行深度测试，从而避免运行不必要的像素着色器程序。

## 更短的汇编不一定更快 Shorter Assembly Is Not Necessarily Faster

现代GPU的结构非常复杂，可能需要通过特殊的方式调度指令来达到最佳的性能。某些情况下编译器会生成额外的指令，使程序运行起来更快：比如使指令能够并行执行，或使指令之间没有依赖关系（指令存在依赖关系意味着后方指令需要等前方指令全部执行完毕才能执行，依赖关系指令可能会导致流水线因为等待个别步骤而完全停止）。

因此，判断着色器的性能不能通过它汇编代码的长度（虽然通常情况下来说，更短的代码意味的更快的速度），而需要测量使用该程序时的帧速率。这需要将所有理论都放在一边，通过量化具体、真实的表现来评估。